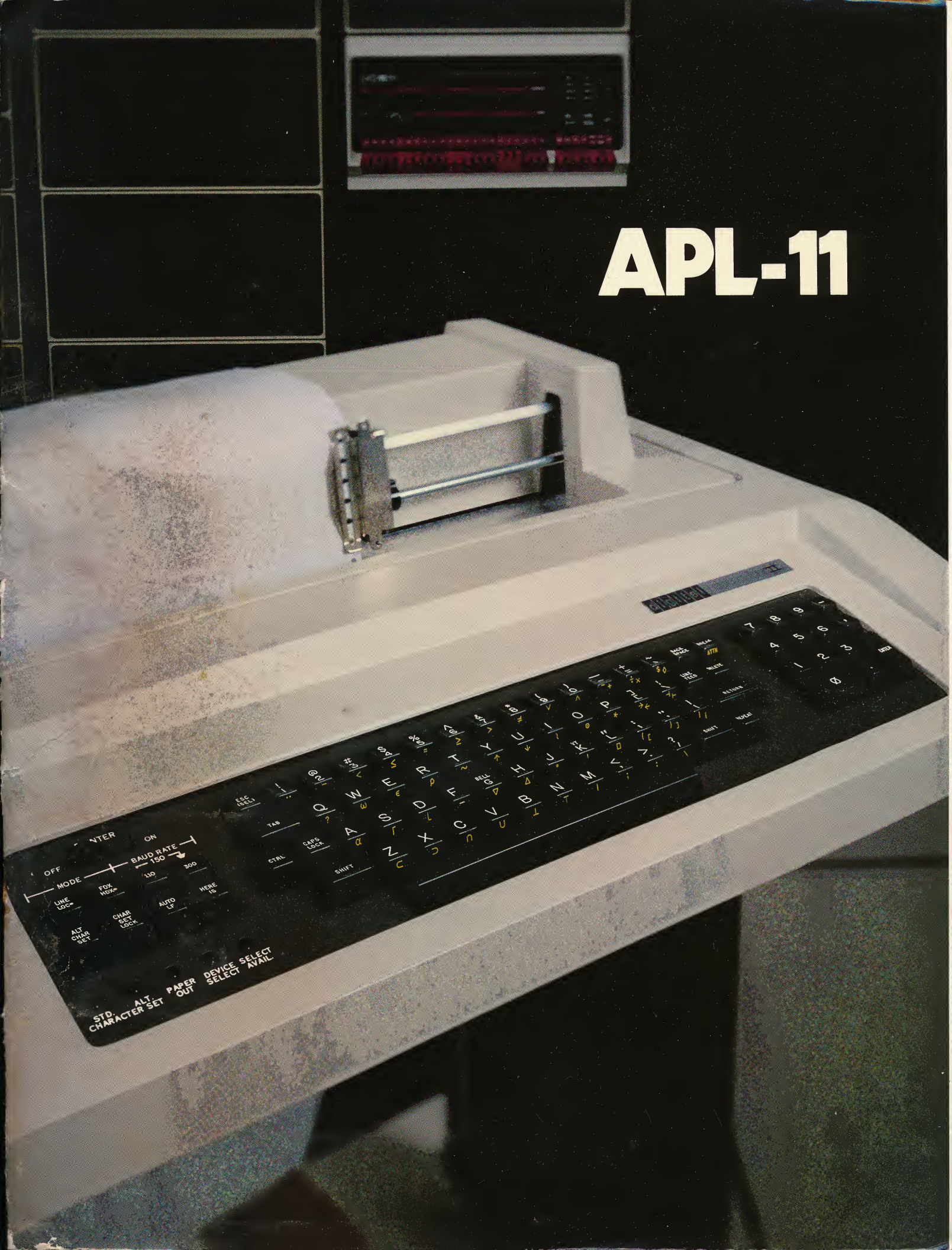
APL-11


OFF
MODE
BAUD RATE
150
110
300
LINE LOC
FOX HOX
AUTO LF
HERE IS
ALT CHAR SET
CHAR SET LOCK
STD. CHARACTER SET
ALT.
PAPER OUT
DEVICE SELECT AVAIL.
TAB
CTRL
CAPS LOCK
SHIFT
RETURN
LINE FEED
DELETE
BACK SPACE
ATTN
BREAK
ENTER

Apr '77

APL

A Flexible Programming Tool for Both Large and Small Computers

One of the keys to APL's growing acceptance as a computer language is its unique ability to be applied successfully to all levels of computer applications. Both large-scale scientific and commercial applications and small-scale interactive computing needs have been implemented efficiently with APL. And it is being used effectively for the whole spectrum of applications in between. Now DIGITAL's APL-11 joins the already well-known DECsystem-10 APL to complete the range of solutions for the APL user.

As a small computer system, DIGITAL's 11V03 personal computer offers much more than just low-cost APL. It is a full-scale disk-based PDP-11 computer system offering FORTRAN IV and BASIC as well as APL. The 11V03 also provides the ability to receive data in real-time for inclusion in sensor-based applications.

For mid-range APL users, DIGITAL's 11/34, 11/55, and 11/70 offer much more than just fast APL execution. The APL user can also take advantage of all the other facilities of PDP-11 systems, including timesharing and real-time operating systems, language processors such as COBOL, FORTRAN, and BASIC, and data handling capabilities such as DBMS-11.

For large-scale APL users, DIGITAL's DECsystem-10 s and DECsystem-20 s offer much more than just a highly developed extended version of APL. The large-scale APL user has access to all the facilities of a production batch and timesharing system, including the development of very large programs and their execution in a virtual memory environment.

In short, DIGITAL offers a complete spectrum of APL capabilities, and complete general purpose computer capability with even the smallest APL system. This brochure describes APL-11, the version of APL that is implemented on all PDP-11 family systems. Descriptions of APL-10, the version of APL that runs on our largest systems, are available from your DIGITAL sales representative.

Introduction

APL is a very concise programming language especially suited for handling array-structured alphanumeric data. APL is used as a general data-processing language as well as a mathematician's tool. The language is flexible enough to solve problems in text-handling and commercial data processing as concisely and as easily as it can solve problems in numerical mathematics and statistics.

APL allows user-defined functions to be expressed with the same syntax used to express primitive language functions. Thus, the user can expand the capabilities of the language to handle the requirements of any application.

Applications

APL is used in engineering, commercial and educational applications. Current applications include: data reduction and analysis, simulation and forecasting, financial modeling, design engineering, electric circuit analysis, engineering analysis, inventory and payroll management, data base manipulation, reservation systems, computer-assisted instruction (CAI), and student education (high school and college level) in programming and the structure of algorithmic processes.

APL on the PDP-11

The APL system is implemented as a language interpreter on the PDP-11. APL can operate on a wide range of hardware processors and has been implemented to run under either of two operating systems: RT-11 or RSTS/E.

The APL run-time system is pre-configured by DIGITAL to match such installation-dependent characteristics as the following:

- PDP-11 processor used
- Operating system (RT-11 or RSTS/E) under which APL will run
- Availability of hardware floating-point processor
- Selection of single-precision or double-precision arithmetic

These characteristics are supplied as system assembly parameters at the time that the system is initially generated.

APL Equipment and Character Set

The user interacts with APL using a typewriter-like terminal or CRT. Two types of terminals are supported by the PDP-11 for use with the APL system.

DIGITAL APL Terminals

Description	Character Set
Any terminal without the APL character set	ASCII
DECwriter II model LA37 with APL option	APL/ASCII

The full APL character set can be represented using a keyboard illustrated in Figure 1. All characters on this keyboard are received and interpreted by APL. Note that letters, numbers, and some of the special characters appear in the conventional keyboard positions.

Terminals Without the APL Character Set

ASCII terminals do not support the use of the special APL characters illustrated in Figure 1. If the user has an ASCII terminal or is operating in console terminal mode on an APL terminal, special APL characters can be represented by keyboard mnemonics. To represent the APL rho symbol (ρ), for example, the user enters the mnemonic .RO. The .GO mnemonic is used to express an APL right-arrow ($\rightarrow$).

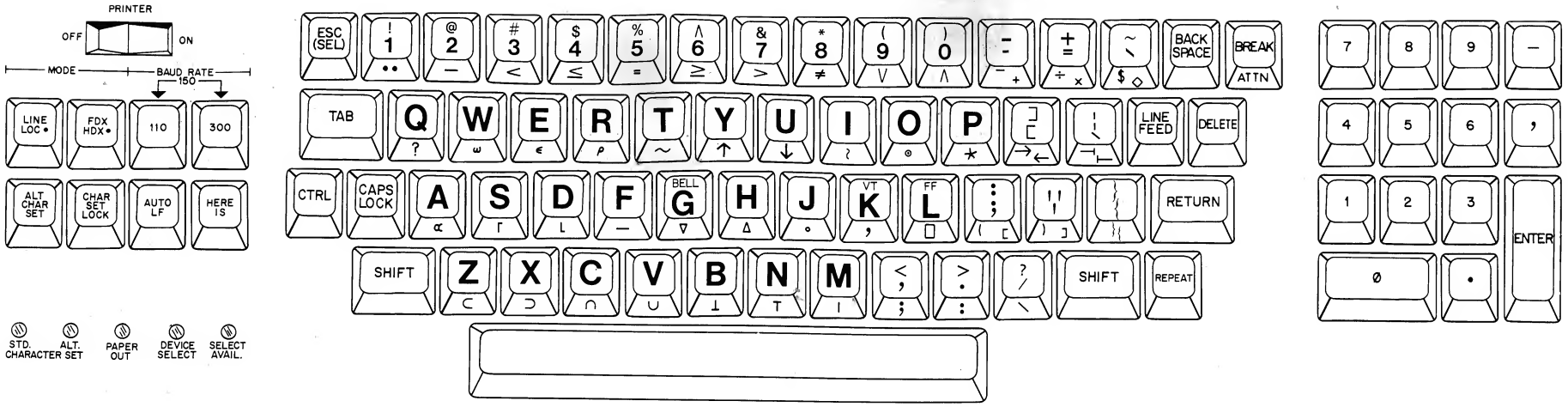


Figure 1. The APL Keyboard (LA37 Terminal)

Language Elements

The APL language system provides functions and operators to manipulate data, and system commands to control the program environment.

Data Structures

Numeric and character data can be structured in a variety of ways. The following data structures are supported by APL:

- scalars
- vectors
- matrices
- arrays with three or more dimensions

A scalar is a single numeric or character value with no dimensions. A character scalar is enclosed in single quotes; for example:

```
enter:        A ← 'C'
          A
returned:     C
```

A vector is a one-dimensional array or character string consisting of any number of values. A numeric vector is entered as a list of values separated by at least one space; for example:

```
enter:        A ← 1 2 3 4
          A
returned:     1 2 3 4
```

A matrix is a two-dimensional array consisting of rows and columns. The user must enter values corresponding to each element of an array, and must also specify the shape of the array. The shape of an array is the number of dimensions which it has and the length of each of these dimensions. For example, a matrix can have six elements arranged as two rows and three columns or three rows and two columns, as illustrated by arrays A and B below.

```
A
1   2   3
4   5   6
B
1   2
3   4
5   6
```

The APL primitive function rho (ρ) is used to specify the shape of a new array or to reshape an existing array. It can also be used to create a null vector, which is extremely useful in certain APL operations. Following is an example of creating a simple matrix with the rho function:

```
enter:    A ← 4 2 ρ 0 1 2 3 4 5 6 7
          A
returned:
          0   1
          2   3
          4   5
          6   7
```

Arrays of three or more dimensions are also supported by APL. There is no intrinsic limit on the number of dimensions that an APL array may have; the only restriction is that the size of the array must not exceed the size of the user's workspace.

Statements

A program consists of one or more lines called statements. There are two types of APL statements:

- assignment statements
- branch statements

Assignment statements include both calculation and input/output operations. Branch statements are used to restart a function or to handle the transfer of control from one part of a program to another. Branch statements are relevant only to user-defined functions.

An APL statement can contain the following components:

- identifiers
- constants
- APL primitive functions
- user-defined functions

Monadic and Dyadic Primitive Functions

APL primitive functions are implemented in two forms: monadic and dyadic. Monadic functions take a right argument and are of the type ÷A (reciprocal), !B (factorial) or ~1 (logical NOT). Dyadic functions take both left and right arguments and are of the type 3+2 (addition), and X=Y (equal). The syntax of the function (i.e., the presence of one or two arguments) determines whether the function is monadic or dyadic. For example, |A is a monadic function used to determine the magnitude or absolute value of the argument A. A|B is a dyadic function used to obtain the residue or remainder available after dividing B by A. The particular operation specified by the | symbol is dependent on the context of the statement.

Primitive Scalar Functions

APL primitive functions are of two types: scalar and mixed. Scalar functions generally take single-number (scalar) arguments and yield scalar results. They are used primarily for basic arithmetic and logical operations, such as addition, exponentiation, maximum value, and logical OR. With a few exceptions, the primitive scalar functions take numeric scalar arguments. The relational functions ($<$, $\leq$, $=$, $>$, $\geq$ $\neq$) can take either character or numeric arguments, but only the equal ($=$) and not equal ($\neq$) primitives may have one character argument and one numeric argument. The logical functions ($\wedge$, V, $\sim$, etc.) must have arguments equal to 0 or 1.

Table 1
Primitive Dyadic Circle Functions

X ○ Y	X	(-X) ○ Y
(1-Y*2)*0.5	0	(1-Y*2)*0.5
SIN Y	1	ARCSIN Y
COS Y	2	ARCCOS Y
TANGENT Y	3	ARCTAN Y
(1+Y*2)*0.5	4	(¯+Y*2)*0.5
SINH Y	5	ARCSINH Y
COSH Y	6	ARCCOSH Y
TANH	7	ARCTANH Y

Tables 2 and 3 summarize the primitive scalar functions available in APL.

Table 2
Primitive Scalar Monadic Functions

FUNCTION	MEANING
+Y	Y
-Y	NEGATIVE OF Y
×Y	SIGN OF Y (¯1,0,1)
÷Y	RECIPROCAL OF Y
*Y	e TO THE Yth POWER
|Y	MAGNITUDE OF Y
⌈Y	CEILING OF Y
⌊Y	FLOOR OF Y
•Y	NATURAL LOGARITHM OF Y
!Y	FACTORIAL Y(FOR INTEGRAL Y GAMMA FUNCTION OF Y+1 FOR NON-INTEGRAL Y)
?Y	A RANDOM INTEGER FROM Y
○Y	π TIMES Y

Table 3
Primitive Scalar Dyadic Functions

FUNCTION	MEANING
X+Y	ADD X TO Y
X-Y	SUBTRACT Y FROM X
X×Y	MULTIPLY X AND Y
X÷Y	DIVIDE X BY Y
X*Y	X TO THE Yth POWER
X|Y	X RESIDUE OF Y
X⌈Y	MAXIMUM OF X AND Y
X⌊Y	MINIMUM OF X AND Y
X•Y	LOG OF Y TO THE BASE X
X!Y	BINOMIAL COEFFICIENT (NUMBER OF COMBINATIONS OF Y THINGS TAKEN X AT A TIME
X○Y	TRIGONOMETRIC FUNCTION (Y IS IN RADIANS)

Extension of Scalar Functions to Arrays

The primitive functions are considered scalar functions because they generally take scalar arguments and yield scalar results. The operations performed by these functions can, however, be extended to arrays. A primitive scalar function is applied to an array on an element-by-element basis. Thus, if the user specifies an addition operation in which both arguments are vectors, the corresponding elements of the vectors are added; for example:

```
enter:      5 8 9 + 6 7 2
returned:   11 15 11
```

The arrays on which the primitive scalar functions operate can be of any dimension. If a dyadic function is being executed, the arrays specified as the arguments of the function must generally have the same number of elements and be the same shape (e.g., a 2-by-3 array is not equivalent to a 3-by-2 array). There is one exception to this rule. If one argument is an array and the other is a scalar or a single-element array, the single value is applied to every element of the array. The following two examples are therefore equivalent.

```
enter:      5 5 5 + 6 7 2
returned:   11 12 7
enter:      5 + 6 7 2
returned:   11 12 7
```


PRINTER
OFF
ON
MODE
BAUD RATE
150
LINE LOC
FDX HDX
110
300
ALT CHAR SET
CHAR SET LOCK
AUTO LF
HERE IS
STD. CHARACTER
ALT. SET
PAPER OUT
DEVICE SELECT
SELECT AVAIL.
ESC (SEL)
TAB
CTRL
CAPS LOCK
SHIFT
BELL

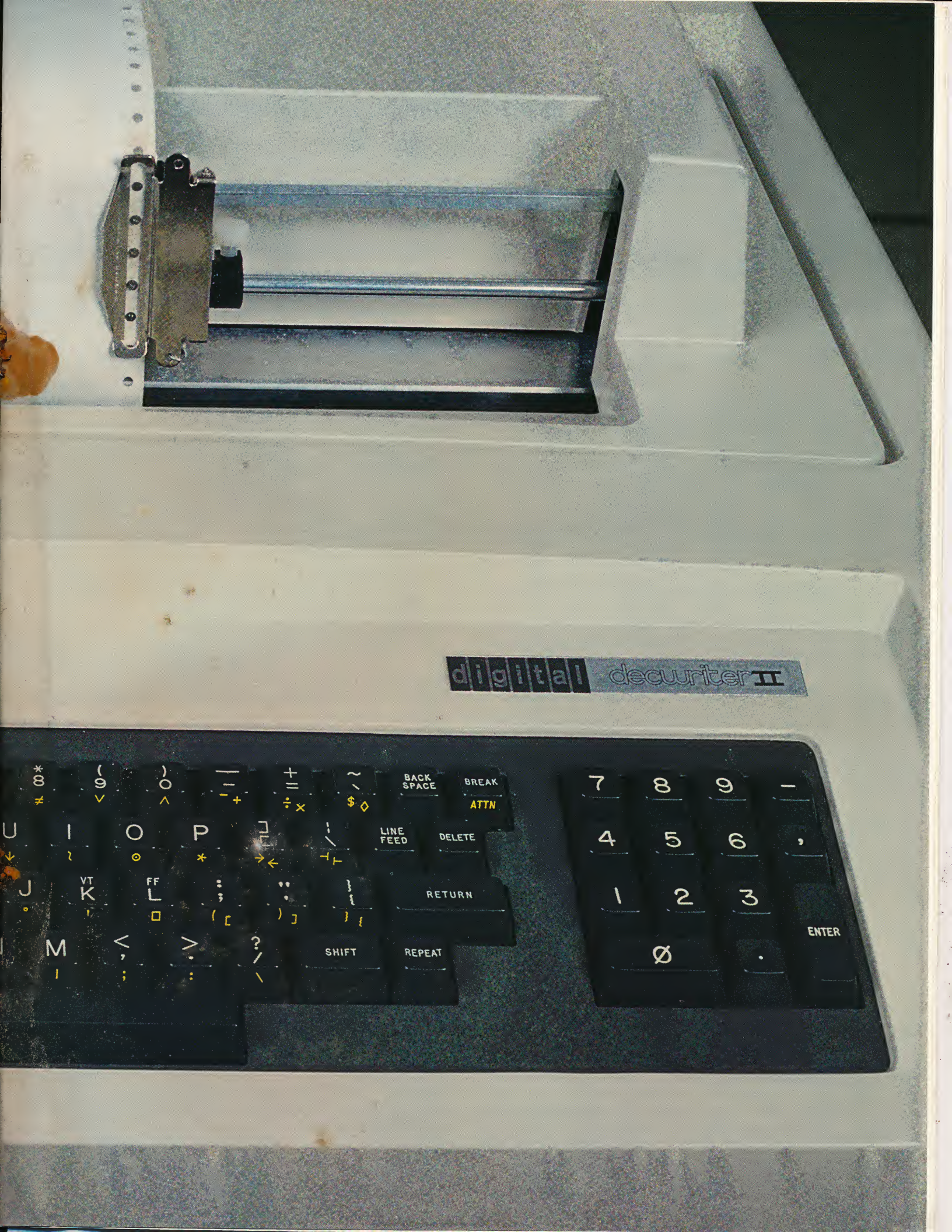

digital decwriter II
BACK SPACE
BREAK
ATTN
LINE FEED
DELETE
RETURN
SHIFT
REPEAT
ENTER

Table 4
Primitive Mixed Functions

FUNCTION	MEANING
⍴Y	RETURN SHAPE OF Y
X⍴Y	RESHAPE Y TO MAKE DIMENSION X
⍳Y	GENERATE THE FIRST Y CONSECUTIVE INTEGERS FROM CURRENT ORIGIN
X⍳Y	FIND THE FIRST OCCURRENCE OF Y WITHIN VECTOR X
,Y	RETURN THE RAVEL OF Y (MAKE Y A VECTOR
X,Y	CATENATE X TO Y ALONG THE LAST DIMENSION OF X
X,[N]Y	LAMINATE X TO Y ALONG THE Nth DIMENSION OF X
X/Y	X (LOGICAL) COMPRESSION ALONG THE LAST DIMENSION OF Y
X/[N]Y	X(LOGICAL) COMPRESSION ALONG THE Nth DIMENSION OF Y
X⌿Y	X (LOGICAL) COMPRESSION ALONG THE LAST DIMENSION OF Y
X\Y	X (LOGICAL) EXPANSION ALONG THE LAST DIMENSION OF Y
X\[N]Y	X(LOGICAL) EXPANSION ALONG THE Nth DIMENSION OF Y
X⍀Y	X(LOGICAL) EXPANSION ALONG THE FIRST DIMENSION OF Y
X↑Y	FOR X>0, TAKE FIRST X ELEMENTS OF Y FOR X<0, TAKE LAST |X ELEMENTS OF Y
X↓Y	FOR X>0, DROP FIRST X ELEMENTS OF Y FOR X<0, DROP FIRST |X ELEMENTS OF Y
⍉Y	TRANSPOSE THE DIMENSIONS OF Y (FOR A MATRIX, EXCHANGE THE ROWS AND COLUMNS)
X⍉Y	TRANSPOSE ARRAY Y ACCORDING TO X
⌽Y	REVERSE ALONG THE LAST DIMENSION OF Y
⌽[N]Y	REVERSE ALONG THE Nth DIMENSION OF Y
⊖Y	REVERSE ALONG THE FIRST DIMENSION OF Y
X⌽Y	ROTATE BY X ALONG THE LAST DIMENSION OF Y
X⌽[N]Y	ROTATE BY X ALONG THE Nth DIMENSION OF X
X⊖Y	ROTATE BY X ALONG THE FIRST DIMENSION OF Y
X⍋Y	GENERATE AN INDEX VECTOR SUCH THAT X[⍋Y] IS IN ASCENDING ORDER
X⍒Y	GENERATE AN INDEX VECTOR SUCH THAT X[⍒Y] IS IN DESCENDING ORDER
X⊥Y	DECODE THE REPRESENTATION OF Y IN NUMBER SYSTEM X
X⊤Y	ENCODE Y IN NUMBER SYSTEM X
?Y	ROLL AN INTEGER SELECTED RANDOMLY IN RANGE 1 THROUGH Y (SCALAR FUNCTION)
X?Y	DEAL X INTEGERS SELECTED RANDOMLY IN RANGE 1 THROUGH Y WITHOUT DUPLICATION
⍎Y	EXECUTE THE CHARACTER STRING Y
X⍎Y	DETERMINE THE MEMBERSHIP OF X IN ARRAY Y
⌹Y	INVERT THE MATRIX Y
X⌹Y	PERFORM MATRIX DIVISION, SOLVE LINEAR EQUATIONS, FIND A LEAST SQUARE SOLUTION

Primitive Mixed Functions

Scalar functions take scalar arguments, yield scalar results, and are extended to arrays on an element-by-element basis. Mixed functions, on the other hand, may take vector arguments and yield scalar or vector results, or may take scalar arguments and yield vector results. In expressing mixed functions for arrays of greater dimensions, it may be necessary to specify the particular coordinate of the array to which the function applies. Table 4 summarizes these functions.

Table 5
Composite Operators

OPERATOR	MEANING
α/Y	THE α REDUCTION ALONG THE LAST DIMENSION OF Y
α/[N]Y	THE α REDUCTION ALONG THE Nth DIMENSION OF Y
α⌿Y	THE α REDUCTION ALONG THE FIRST DIMENSION OF Y
Xα.αY	GENERALIZED INNER PRODUCT OF X AND Y
X∘.αY	GENERALIZED OUTER PRODUCT OF X AND Y

Relational Functions

In APL, the relational functions (<, ≤, =, >, ≥ ≠) return results; they are not simply comparison operators. An expression of the form A<B yields a result of 1 true if A is less than B and 0 false if A is greater than or equal to B. For example:

```
enter:      9>6
returned:   1
enter:      4>6
returned:   0
enter:      'C'>'A'
returned:   1
```

Table 6 summarizes these functions.

Table 6
Logical Functions

FUNCTION	MEANING
X<Y	X LESS THAN Y
X≤Y	X LESS THAN OR EQUAL TO Y
X=Y	X EQUAL TO Y
X≥Y	X GREATER TO OR EQUAL TO Y
X>Y	X GREATER THAN Y
X≠Y	X NOT EQUAL TO Y
X∧Y	X AND Y
X∨Y	X OR Y
X⍲Y	X NAND Y (NOT BOTH X AND Y)
X⍱Y	NEITHER X NOR Y
~Y	NOT Y

Input/Output Operations

Input/output statements are a special variety of assignment statements. In APL, input and output operations are generally expressed by the special quad operator, □.

Quad Input Mode

The most basic form of input to APL is called evaluated input. In this mode, the value entered by the user is assigned to the variable to the left of the specification arrow. In the following example:

```
K←□
18
```

The K variable takes on the value 18 entered by the user at the terminal.

If a quad symbol appears anywhere in an APL statement except immediately to the left of a left-arrow, input will be accepted from the terminal as in the following:

```
A←3x□+5
7
```

Here the value of A becomes 36=3x(7+5).

Quote-Quad Input Mode

A version of the quad operator called the quote-quad operator (⍞) is used especially for the input of character data. An example of quote-quad mode is shown below.

```
X←⍞
THAT'S AMAZING
X
THAT'S AMAZING
```

Unlike evaluated input, quote-quad input allows character strings to be entered without explicit quote characters. When APL encounters a ⍞ symbol, it positions the carriage at the left margin and accepts the data entered by the user up to the next carriage return as a character string. If a single character is entered, APL treats it as a literal scalar; a string is stored as a literal vector. If the user enters only a carriage return, APL treats this input as a vector of length zero. This is significantly different from the handling of empty input in evaluated input mode, in which APL rejects the input and waits for the user to reenter it.

Quad-Del Input Mode

A special version of the quad operator, the quad-del operator (⍁) is used to input characters exactly as typed by the user. No special editing of APL characters is performed. The backspace, for example, is treated as a special character, and an overstrike symbol is not created. The following statements illustrate this difference between quad-del and quote-quad modes.

```
      X←⍞
⍞:    ϕA
      ρX
      4
      X←⍁
⍁:    ϕA
      ρX
      2
      ρ'ϕA'
      2
```

Normal Output Mode

If a quad symbol appears immediately to the left of a left arrow, the value of the expression to the right of that specification arrow is output. Terminal output can also be accomplished simply by entering the name of the variable whose value is to be displayed. For example, the APL statement:

```
□←B
```

using the quad symbol is equivalent to the statement:

```
B
```

since both have the effect of displaying the value of B.

The quad output mode is especially helpful when an APL statement consists of multiple specifications. For example:

```
X←15-□←3+4
7
```

Here the quantity 7 is assigned to the quad operator and displayed. The value of X is computed (its value is 8) but not displayed.

Table 7
Keyboard I/O Operators

OPERATOR	MEANING
X←□	QUAD (EVALUATED) INPUT FROM KEYBOARD
X←⍞	QUOTE-QUAD (CHARACTER) INPUT FROM KEYBOARD, UP TO BUT NOT INCLUDING CARRIAGE RETURN
X←⍁	QUAD-DEL (UNEDITED) INPUT FROM KEYBOARD
⍬←X	QUAD OUTPUT (DISPLAY VALUE OF X)

Heterogeneous Output Mode

It is often desirable to mix character and numeric data on the same output line. Mixed output lines of this kind are called heterogeneous output. The APL user requests heterogeneous output simply by entering a series of values or expressions, separated by semicolons, in the order in which they are to appear; the values can be parenthesized. The output displayed as a result of this specification contains no carriage returns, except where required by the data.

Communication with the System

There are several ways in which the user can communicate with the APL system to change system parameters, determine hardware or operational characteristics, and modify workspace parameters. The system commands facilitate many of these system operations, and the I-beam functions allow APL users to communicate with the system from within the APL language itself.

System Commands

System commands provide a means of communicating with the APL system and controlling the operational environment in which an APL session is conducted. System commands allow the user to examine or change the state of the system in such ways as the following:

- Clear, identify, or save the active workspace
- Load or delete a workspace from a secondary storage device
- List variable and function names
- Display the status of functions and variables in the workspace
- Set the index origin, maximum number of significant digits, output line width, and comparison tolerance

I-beams

I-beams are APL functions used to communicate with the APL system to change user workspace characteristics and to report statistics about the workspace and the APL system. Unlike system commands, these functions are subject to the APL language syntax and rules of function definition. They can be included in user functions and defined in conjunction with other language operations.

There are two types of I-beam functions. The first category consists of functions used to return workspace and system information. The following are examples of information returned by the I- beams in this category.

- Symbol table size and remaining available space
- Date and time of day
- System job number, user's project-programmer number, and terminal character set
- Line numbers in the state indicator
- Precision of APL version
- Values of system assembly parameters

Some of these I-beams report in general system characteristics (e.g., date) and others return information relevant only to the particular user's workspace and session.

The second I-beam category consists of functions used to perform system actions and change workspace parameters. The following are examples of actions performed by the I-beams in this category:

- Turning error displays for the execute operator on and off
- Terminating the APL session
- Selecting the terminal character set
- Changing the random number sequence

Writing APL Programs

APL statements can be executed in either of two modes:

- Immediate mode, in which functions, statements, and expressions entered by the user are evaluated and executed immediately
- Function-definition mode, in which the user can construct a program consisting of APL statements, and name and save the program

The APL user can conveniently shift from one mode to the other by typing a mode-transfer 'del' (∇) symbol. The syntax of the APL language itself is identical in both modes. Some special symbols are defined for ease of editing in function-definition mode.

Defining the Function

In the APL language, a program is implemented as a user-defined function. A user-defined function can include both APL primitive functions and other user-defined functions. The user develops a program in APL function-definition mode. Once developed, that program may be used with the convenience of a primitive function.

A function is constructed in two parts: a function header and a function body. The function header defines the name of the function, the syntax of the function call, and any variables defined to be local to the function. The function body consists of a number of program statements that define the actions to be performed by the function when it is executed.

Editing the Function

A function definition can be altered by the user in a variety of ways. Definition lines can be added, deleted, displayed, and changed, and the function header can be altered. The APL statements that make up a function definition are neither executed nor checked for syntactic validity when entered by the user. In function-definition mode, the user simply enters statements, edits them to correct obvious mistypings and inconsistencies, and saves them for future use.

Executing the Function

The process of defining a function associates the function header provided by the user with the statements entered as the function body. When the user decides to execute the defined function, he uses the function as he would a primitive APL function. The information provided in the function header specifies the number of arguments to be supplied in the function call and determines whether or not a value will be returned.

Debugging the Function

Function execution is suspended before normal completion if an error occurs or if a stop vector is set. When execution is suspended, the name of the suspended function and the line number of the statement that would have been executed next are displayed. APL then awaits input in immediate mode. The user can perform any other APL operations at this time. The user can resume execution after fixing the problem, and observe function nesting.

The Trace Vector

For debugging purposes, the user may find it helpful to obtain an automatic display of the intermediate results of function execution. As a program tracing aid, the values computed by one or more function statements can be output each time those statements are executed. To establish a trace for lines 4, 6, and 7 of function F, the user includes the following statement in the function definition:

```
T∆F←4 6 7
```

For each execution of the specified line numbers, this command causes the following information to be displayed, in the order shown:

- function name
- line number
- final value returned by the statement

The Stop Vector

APL allows the user to suspend execution of a function from within the function itself. A stop control vector is available, with a syntax similar to that of the trace vector. The stop can be used to suspend function execution just before execution of one or more specified statements. To cause function F to be suspended before executing line 12 and line 19, the user includes the following statement in the function definition:

```
S∆F←12 19
```

For each suspension, this command causes the function name and line number that was about to be executed to be displayed. To disable the stop vector for function F, the following specification is supplied:

```
S∆F←⍳0
```

After function execution has been suspended by the stop control vector, the system is in the normal suspended state. Execution can be resumed by specifying a branch to the desired line number.

The stop control vector can be set from within a function to cause suspension only under certain circumstances. Editing a line for which a stop vector has been defined causes the stop vector to be disabled for that line.

Environments

The RT-11 and RSTS/E operating systems provide APL users with many of the standard features of the PDP-11 real-time and timesharing environments. When configured for single-user access under RT-11, the APL interpreter uses four overlay segments and requires about 14K words of memory. When configured for use with RSTS/E, several users can simultaneously access the APL system. The APL interpreter can be configured as a RSTS/E run-time system. Each user shares the reentrant interpreter code, and only the user's workspaces are swapped. The APL run-time system uses approximately 16K words of memory.

APL-11 can be used on a variety of PDP-11 processors. It has been optimized to make efficient use of systems that offer hardware floating-point processors; for example, the PDP-11/34, 11/45, 11/55 and 11/70. However, APL can also be configured for use with processors that do not have floating-point processors. If a hardware processor is not available, a software floating-point package will be assembled with the APL interpreter to simulate the floating-point hardware.

APL-11 can be generated to perform either single-precision or double-precision arithmetic. Single precision provides an accuracy of approximately seven digits, and double precision offers an accuracy of about 16 digits. I-beam 37 can be used to determine the precision of a particular APL system. At the time that APL is initially configured, the character set available on user terminals at the installation can be specified.

Workspaces

An APL workspace is a buffer in the user's memory area which stores the functions, variables, values, and temporary results obtained while executing APL statements. Using the APL system commands, workspaces can be saved, retrieved, and erased in the same manner as any other file. They can be stored on a variety of PDP-11 devices, including disk, magnetic tape, DECtape, and floppy disk.

A workspace can be saved in either memory-image or ASCII format. Workspaces saved in ASCII form can be created and edited with any DEC editor. This is an important feature not found on many APL systems.

There may be several workspaces stored in the user's disk area. The workspace currently available to the user is known as the active workspace. The maximum APL workspace depends upon the operating system and the amount of memory in the system. In an RT-11 system with 28K words of memory, the workspace may be approximately 24,000 bytes. Under RSTS/E, the maximum workspace is even larger.

File System

The APL-11 file system allows the APL user to access data and program files on a variety of system devices, including disk, DECtape, magnetic tape, and floppy disk. The file system is implemented as an integral part of the APL language and provides an interface to the RSTS/E and RT-11 operating systems.

The APL file system support is provided by:

- System commands for use in assigning, creating, closing, reading, writing, and renaming files
- File operators for byte pointer, input, and output functions

Table 8
File I/O Operators

OPERATOR	MEANING
CHANNEL-NUMBER ↑ [TYPE] N	SET FILE POINTER
CHANNEL-NUMBER ← [TYPE] N	FILE INPUT
CHANNEL-NUMBER → [TYPE] DATA	FILE OUTPUT

In the APL-11 file system, input and output functions can be specified for files associated with any of 13 channels, one of which is reserved for use by the user's terminal. The ASSIGN file system command is used to associate an existing file with a channel. CREATE is used to create new files on specified channels by allocating space for them.

Two types of files are supported by the APL system:

- ASCII sequential
- Random access

APL ASCII sequential data files can be read and written sequentially by any other RT-11 or RSTS/E language processor (e.g., BASIC, FORTRAN, or MACRO). In addition, APL programs can create and read any standard ASCII files. Because APL workspaces can be read and stored in ASCII format, the file system can be used to save, retrieve, and manipulate these workspaces.

The APL system also supports the use of random-access files. The system treats a file as random-access memory, and the user can access directly any byte in the file by specifying the individual byte or value to be read or written. APL can access random files in any format, created with almost any language processor or system. For example, random-access mode can be used to read and write FORTRAN direct-access data files.

digital

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone: (617)897-5111—SALES AND SERVICE OFFICES: UNITED STATES—ALABAMA, Huntsville • ARIZONA, Phoenix and Tucson • CALIFORNIA, El Segundo, Los Angeles, Oakland, Ridgecrest, San Diego, San Francisco (Mountain View), Santa Ana, Santa Clara, Stanford, Sunnyvale and Woodland Hills • COLORADO, Englewood • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington (Lanham, MD) • FLORIDA, Ft. Lauderdale and Orlando • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago (Rolling Meadows) • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans (Metairie) • MARYLAND, Odenton • MASSACHUSETTS, Marlborough, Waltham and Westfield • MICHIGAN, Detroit (Farmington Hills) • MINNESOTA, Minneapolis • MISSOURI, Kansas City (Independence) and St. Louis • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Metuchen and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Albany, Buffalo (Cheektowaga), Long Island (Huntington Station), Manhattan, Rochester and Syracuse • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland (Euclid), Columbus and Dayton • OKLAHOMA, Tulsa • OREGON, Eugene and Portland • PENNSYLVANIA, Allentown, Philadelphia (Bluebell) and Pittsburgh • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville and Nashville • TEXAS, Austin, Dallas and Houston • UTAH, Salt Lake City • VIRGINIA, Richmond • WASHINGTON, Bellevue • WISCONSIN, Milwaukee (Brookfield) • INTERNATIONAL—ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth and Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro and Sao Paulo • CANADA, Calgary, Edmonton, Halifax, London, Montreal, Ottawa, Toronto, Vancouver and Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • FINLAND, Helsinki • FRANCE, Lyon, Grenoble and Paris • GERMAN FEDERAL REPUBLIC, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nuremburg, Stuttgart and West Berlin • HONG KONG • INDIA, Bombay • INDONESIA, Djakarta • IRELAND, Dublin • ITALY, Milan, Rome and Turin • IRAN, Tehran • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Utrecht • NEW ZEALAND, Auckland and Christchurch • NORWAY, Oslo • PUERTO RICO, Santurce • SINGAPORE • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • UNITED KINGDOM, Birmingham, Bristol, Epsom, Edinburgh, Leeds, Leicester, London, Manchester and Reading • VENEZUELA, Caracas •

PRINTED IN U.S.A. EK-[illegible]-TM-004 015